A Scalable Parallel Poisson Solver in Three Dimensions with Infinite-Domain Boundary Conditions

Peter McCorquodale
Phillip Colella
Applied Numerical Algorithms Group
Lawrence Berkeley National Laboratory
Berkeley, CA USA
{pwmccorquodale,pcolella}@lbl.gov

Gregory T. Balls
Scott B. Baden
Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114 USA
{gballs,baden}@cs.ucsd.edu

28 February 2005

Abstract

We present an elliptic free-space solver that offers vastly improved scalability over a previous variant of the algorithm. The new algorithmic changes enable us to scale up to 4096 processors of an IBM SP system, and we are planning to port the solver to Blue Gene L. The solver employs a method of local corrections that avoids the need for costly communication, while retaining scalability of the method. Communication costs are small, on the order of a few percent of the total running time. The numerical overheads incurred are independent of the number of processors for a wide range of problem sizes. The solver currently handles for infinite-domain (free space) boundary conditions, but may be reformulated to accommodate other kinds of boundary conditions as well.

1 Introduction

Elliptic solvers for free space problems often scale poorly owing to the difficulty in treating the infinite-domain boundary conditions. While some of the underlying communication could be masked by overlapping it with useful computation [15, 16, 4, 3, 2], the approach is ultimately non-scalable, as the total cost of communication grows with the size of the problem (**ALL: WE NEED TO CHECK THIS, BUT THAT's MY INTUITION in handling free space BCs**). We present an alternative approach that algorithmically reformulates the solution, reducing the amount data communicated at the expense of additional computation. In particular, we use a local corrections strategy that divides the solution into low and high resolution components, and uses the low resolution component to reduce the amount of communication. The resultant algorithm employs a fixed number of communication and computation steps. The added computational

overheads are purely local work, and the cost is independent of the number of processors over a wide range of problem sizes. Our algorithm exploits elliptic regularity, an approach which is also employed by the fast multipole method [10], the method of local corrections for particle methods [1], the finite element of Bank and Holst [8], and the two-dimensional method of local corrections for free space problems[5, 7]. Results presented by Holst [11] include a rigorous proof that these types of algorithms can produce accurate results with little communication.

We have previously reported early results with our solver, originally called Scallop. While that solver enabled us to avoid high communication overheads, computation of the infinite-domain boundary conditions became a bottleneck and numerical overheads hampered scalability beyond 1,024 processors of an IBM SP system with POWER3 processors. We discuss algorithmic enhancements that greatly reduce the computational overheads and diminish the total running time required to reach a solution. Our new solver, now called Chombo-MLC, solves elliptic partial differential equations with infinite-domain (free space) boundary conditions, which are especially useful for certain astrophysics problems. While Chombo-MLC can be altered to handle other boundary conditions, we focus here on on the infinite-domain case. For purposes of simplification, we restrict the discussion to the Poisson equation.

Unlike domain decomposition methods such as [14] which require multiple iterations between the local and nonlocal descriptions, Chombo-MLC does not perform repeated iterations between coarse and fine levels or several communication steps. Chombo-MLC reaches a solution to the Poisson equation in three steps and communicates data only twice. First, coarse grid data are communicated to generate a global coarse grid charge field. Second, coarse and fine boundary condition data are communicated once among neighboring regions. These communication costs are low in practice–on the order of a few percent–and come at the expense of computational (numerical) overhead. We show that the extra computation involved is reasonable, and significantly, that the computational overhead is independent of the number of processors for a wide range of problem sizes. As a result, we are able to demonstrate scalability on up to 4096 processors of an IBM SP system, and we plan future computations on thousands of processors.

Our contribution to our prior work with Scallop comes in two parts. First, we greatly improve the speed of our serial infinite-domain solution by using the fast multipole method to calculate the necessary boundary conditions. Second, we now calculate coarse grid values necessary for the method of local corrections simultaneously with the initial local solutions, also using the fast multipole method. By calculating coarse grid values in the correction radius in this way, we are able to scale up to thousands of processors with much lower computational overhead.

Chombo-MLC is representative of a class of algorithms that employ sophisticated numerical techniques to reduce communication costs. The techniques in turn require an appropriate software infrastructure to manage the underlying details, in particular the bookkeeping. To this end, the original version of Scallop was built with the KeLP programming system [9], a framework for implementing scientific applications on distributed memory parallel computers. KeLP provides geometric and communication abstractions that facilitated the development of Scallop without sacrificing performance. The current implementation of our solver is part of the Chombo package **reference?**, which provides many of the same geometric and communications abstractions as KeLP as well as infrastructure for adaptive mesh refinement.

2 Preliminaries

The equation we solve is the Poisson equation in three dimensions with a charge distribution ρ with compact support, i.e. the charge is only nonzero in a finite region of space. Specifically, we seek

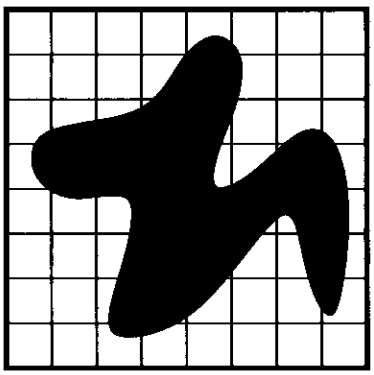

Figure 1: Discretization. The outer boundary of the grid shown here corresponds to Ω^h. The regular mesh, with spacing h represents the uniform discretization of Ω^h. The charge field is non-zero within the colored region shown, also called the support of the charge ρ. The support of ρ lies completely within Ω^h.

the solution ϕ to

$$\triangle\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = \rho(x,y,z)$$

which has far-field behavior characterized by

$$\phi = -\frac{R}{4\pi|\vec{x}|} + o\left(\frac{1}{|\vec{x}|}\right), \qquad |\vec{x}| \to \infty,$$

where R is the total charge:

$$R = \int_\Omega \rho(\vec{x})\, d\vec{x},$$

and the region Ω contains the support of the charge ρ.

For many engineering calculations, methods which are accurate to $O(h^2)$ provide a good balance between accuracy and work required. We seek a solution which is accurate to $O(h^2)$ over the discretized computational domain Ω^h, where h is the uniform discretization, as illustrated in Figure 1. This computational domain corresponds to the index set of the discrete solution ϕ^h, i.e. the indices of the underlying discrete mesh.

Since our goal is to solve the problem on parallel processors, we partition Ω^h into a set of disjoint subdomains Ω_k^h:

$$\Omega^h = \bigcup_k \Omega_k^h.$$

Our method entails solving local problems on each of the Ω_k^h in parallel, as well as on a single coarsened global mesh Ω^H. The spacing of this coarsened mesh is $H = Ch$, where C is a specified *coarsening factor.*

We choose the domain Ω^h to be a rectangular region, $\Omega^h = [\vec{l}, \vec{u}]$, where $\vec{l}$ and $\vec{u}$ are the integer vectors corresponding to the lower and upper corners of the region. The coarsened domain is then defined as

$$C(\Omega^h, C) = \Omega^H = [\lfloor \vec{l}/C \rfloor, \lceil \vec{u}/C \rceil]$$

where the operators $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ represent the *floor* and *ceiling* operators, respectively.

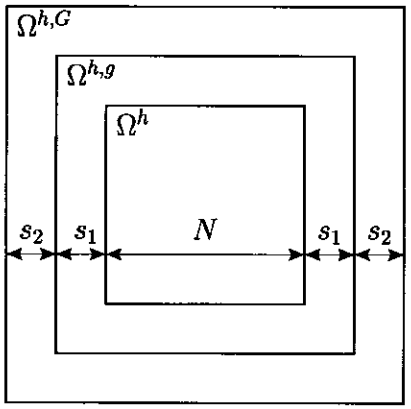

Figure 2: Domains used in James's algorithm, showing lengths in index space.

Because our meshes are node-centered, the points of Ω^H map directly onto corresponding points in Ω^h, and no averaging is required to coarsen the mesh data. Thus, we can coarsen the mesh by sampling the mesh without having to interpolate. In particular, we coarsen a fine grid representation using the *sample* operator $\mathcal{S}^H$: for each point $\vec{x}_C$, we can find the coarse grid value $\psi^H(\vec{x}_C)$ (where ψ^H has grid spacing H) by finding the fine grid point $\vec{x}$ at the corresponding position in ψ^h (with grid spacing $h = H/C$):

$$\psi^H(\vec{x}_C) = (\mathcal{S}^H(\psi^h))(\vec{x}/C) = (\psi^h)(\vec{x})$$

For the discussion that follows, we also need one more bit of notation. The *grow* operation extends or shrinks an index domain by a uniform amount in each direction. If $\Omega^h = [\vec{l}, \vec{u}]$ (where $\vec{l} = (\mathtt{l}_x, \mathtt{l}_y, \mathtt{l}_z)$ and $\vec{u} = (\mathtt{u}_x, \mathtt{u}_y, \mathtt{u}_z)$), we define *grow* as

$$grow(\Omega^h, g) = [\vec{l} - (g, g, g), \vec{u} + (g, g, g)].$$

When $g < 0$, grow returns a shrunken domain.

3 The Method

Our domain decomposition method is built upon a method for solving single-processor infinite-domain Poisson problems, as described previously in [6]. We will summarize the single-grid algorithm first, and then describe the domain decomposition algorithm.

3.1 A Serial Infinite-Domain Poisson Solver

Following the approach described in [12] and [13], we are able to calculate a solution to the Poisson equation with infinite-domain boundary conditions in four steps, using two solution grids. Given a charge on a grid Ω^h, the first solution grid, referred to here as the *inner grid* or $\Omega^{h,g}$, is s_1 points larger in each dimension than Ω^h. The second grid, called the *outer grid* or $\Omega^{h,G}$, is larger still, expanded by s_2 points from $\Omega^{h,g}$ in each dimension. The relationships among these three grids are depicted in Figure 2. The four steps required to calculate the solution are as follows.

1. Find the solution to the Poisson equation on the inner grid, $\Omega^{h,g}$, using Dirichlet boundary conditions.

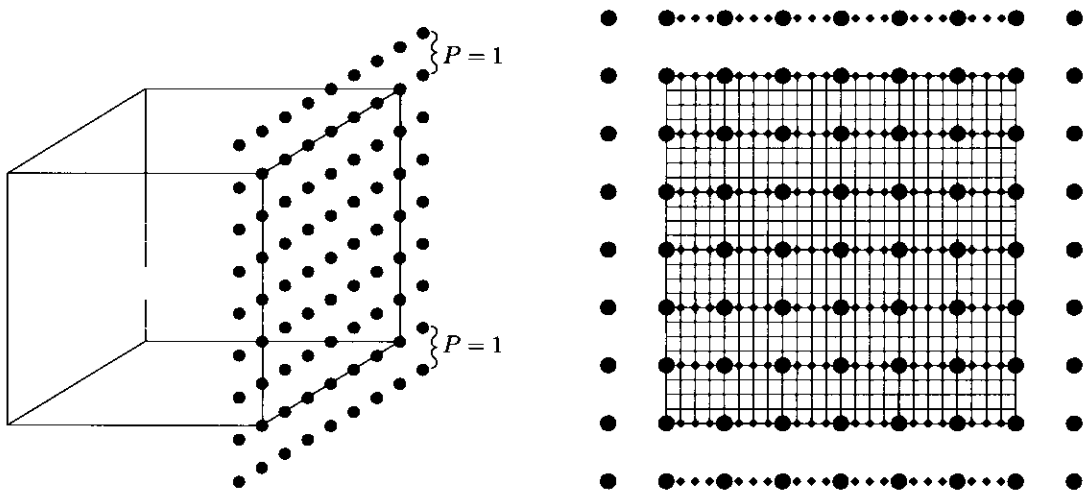

Figure 3: In step 3, multipole moments are calculated for each patch on $\partial\Omega^{h,g}$, such as the one shaded in red. The multipole expansions are then evaluated at the coarse points of $\partial\Omega^{h,G}$, plus an additional layer of width P, indicated with blue circles for one face. These evaluations are interpolated to the fine points of $\partial\Omega^{h,G}$, located at intersections of the black lines, using two passes. The evaluation points of the first pass are shown as green diamonds.

2. Calculate a charge, q, along the inner grid boundary equal to the normal derivative of initial solution at the inner grid boundary.

3. Calculate boundary conditions at each point on the outer grid by numerically integrating the effect of the charge at the inner grid boundary:

$$g(\vec{x}) = \int_{\partial\Omega^{h,g}} G(\vec{x} - \vec{y}) q(\vec{y}) dA_{\vec{y}}.$$

4. Find the solution to the Poisson equation on the outer grid, $\Omega^{h,G}$, using the boundary conditions, g, just calculated.

Our approach here is identical to the approach described previously in [6] except in the way the integration is performed in step 3. Previously, we integrated the charge from the inner grid onto a coarsened version of the outer grid, with mesh spacing $H = h/O(\sqrt{N})$, and interpolated from the coarse grid to find necessary values on $\partial\Omega^{h,G}$. Our straightforward integration required $O(N^3)$ work but significantly took more time to compute than the Dirichlet solutions on the inner and outer grids.

In our current implementation, we perform the integration required for the boundary calculation using the fast multipole method (FMM). Each face of $\Omega^{h,g}$ is divided into patches of $C \times C$ points. We then calculate the multipole moments of the charge up to order M on each patch. On each face of $\Omega^{h,G}$, for points on a mesh coarsened by C in each dimension and expanded by a coarse layer of points of width P, we add up the evaluations of multipole expansions due to all the faces of $\Omega^{h,g}$. Finally, we interpolate polynomially, one dimension at a time from the coarse mesh values to the remaining fine mesh points on the face. (See Figure 3.)

Choosing $C = \sqrt{N}$ provides sufficient accuracy for the solution and allows the integration step to be completed in $O((M^2+P)N^2)$ work. The values of M and P are chosen with regard to accuracy

N	C	s_2	N^G	N^G/N
16	4	6	28	1.75
32	8	12	56	1.75
64	8	12	88	1.38
128	12	20	168	1.31
256	16	24	304	1.19
512	24	44	600	1.17
1024	32	48	1120	1.09
2048	48	80	2208	1.08

Table 1: Values of the coarsening factor, C, annulus thickness, s_2, and resulting expanded grid size, N^G, for various input grid sizes N. The ratio of N^G/N decreases for increasing N.

requirements and are independent from N, so for a given degree of accuracy, the integration step requires $O(N^2)$ work.

We should also note the constraints required on s_1 and s_2, the spacing between the grids Ω^h, $\Omega^{h,g}$, and $\Omega^{h,G}$. We have found that setting $s_1 = 0$ has only small effects on the accuracy of our solutions and doing so allows us to minimize the size of the solution grids. Convergence requirements of the multipole method force us to choose s_2 with more care, however. In order for the multipole expansions from a patch to converge, the distance from a patch center on $\partial\Omega^{h,g}$ to the points on $\partial\Omega^{h,G}$, on which the expansion is evaluated, should be at least twice the radius of the patch. Here we define the radius of a patch as the maximum distance from the patch center to any point on the patch. Recall that we chose our patches to be C fine grid points square. Thus our patches have a radius of $Ch/\sqrt{2}$, and the distance requirement becomes $s_2 h \geq 2(Ch/\sqrt{2})$. We also need the number of cells along the length of $\Omega^{h,G}$ to be divisible by C. Combining these two requirements, we arrive at the following formula for s_2:

$$s_2 = \frac{C}{2}\lceil 2\sqrt{2} + \frac{N}{C}\rceil - \frac{N}{2}. \tag{1}$$

In order to demonstrate the effect of these requirements, we show in Table 1 the necessary values of s_2 for grid sizes, N, ranging from 16 to 2048 by powers of 2. Values of C are chosen to be close to the square root of N but also multiple of four. Note that the ratio of N^G (the length of $\Omega^{h,G}$) to N decreases as N increases. This implies that overhead will be smaller for solutions to larger infinite-domain problems.

3.2 Domain Decomposition

The domain decomposition algorithm described here is a finite-difference analogue of Anderson's method of local corrections [1]. Our algorithm consists of three computational steps interspersed by two communication steps, as described previously in [6].

1. INITIAL LOCAL SOLUTION. We calculate a local infinite-domain solution on each local subdomain, k, augmented with an overlap region:

$$\triangle_{19}\phi_k^{h,initial} = \rho_k^h \text{ on } grow(\Omega_k^h, s + Cb).$$

and construct a coarsened version of the solution, $\phi_k^{H,initial}$, by sampling:

$$\phi_k^{H,initial} = \mathcal{S}^H(\phi_k^{h,initial}) \text{ on } grow(\Omega_k^H, s/C + b).$$

Here s is a correction radius, C is the coarsening factor, and b is the width of a layer for polynomial interpolation to be used in step 3. The $\triangle_{19}$ operator represents the Laplacian calculated with a 19-point stencil of nearest neighbor points. The error characteristics of the 19-point stencil are essential for maintaining $O(h^2)$ accuracy in the overall algorithm when combining the effects of coarse and fine grid data later on.

2. GLOBAL COARSE SOLUTION. We couple the individual local solutions by solving another Poisson equation on a coarsened mesh covering the entire domain. We first construct coarsened local charge fields:

$$R_k^H = \begin{cases} \triangle_{19}\phi_k^{H,initial} & \text{on } grow(\Omega_k^H, s/C - 1), \\ 0 & \text{otherwise} \end{cases}$$

and then sum up these charge fields to form a global coarse representation of the charge:

$$R^H = \sum_k R_k^H.$$

Then we solve

$$\triangle_{19}\phi^H = R^H \text{ on } grow(\Omega^H, s/C + b)$$

with infinite-domain boundary conditions.

3. FINAL LOCAL SOLUTION. We solve

$$\triangle_7\phi_k^h = \rho_k^h on \Omega_k^h$$

with Dirichlet boundary conditions on $\partial\Omega_k^h$:

$$\phi_k^h(\vec{x}) = \sum_{k':\vec{x}\in grow(\Omega_{k'}^{h,initial},s)} \phi_{k'}^h(\vec{x}) + \mathcal{I}\left(\phi^H(\vec{x}) - \sum_{k':\vec{x}\in grow(\Omega_{k'}^h,s)} \phi_{k'}^{H,initial}(\vec{x})\right)$$

where $\mathcal{I}$ is the same interpolation operator used in the serial infinite-domain Poisson solver. Figure 4 depicts the regions from which data are taken to set boundary conditions on a face.

Note that the algorithm does not require fine grid data at all points in $grow(\Omega_k^h, s+rb)$. Specifically, the algorithm does not need fine grid data outside of $grow(\Omega_k^H, s/r + b)$. In order to complete steps 2 and 3, we only need the following data from step 1:

- the solution at coarse grid values, $\phi_k^{H,initial}$, on $grow(\Omega_k^H, s/C + b)$, necessary for step 2, and
- the solution at fine grid values, $\phi_k^{h,initial}$, on the faces of $grow(\Omega_k^h, s)$.

To ensure accuracy of the method, we need $s = 2C$.

As before, communication is only required in two phases: first, in constructing the global coarse charge field, and second, to set the boundary conditions for the final local solution.

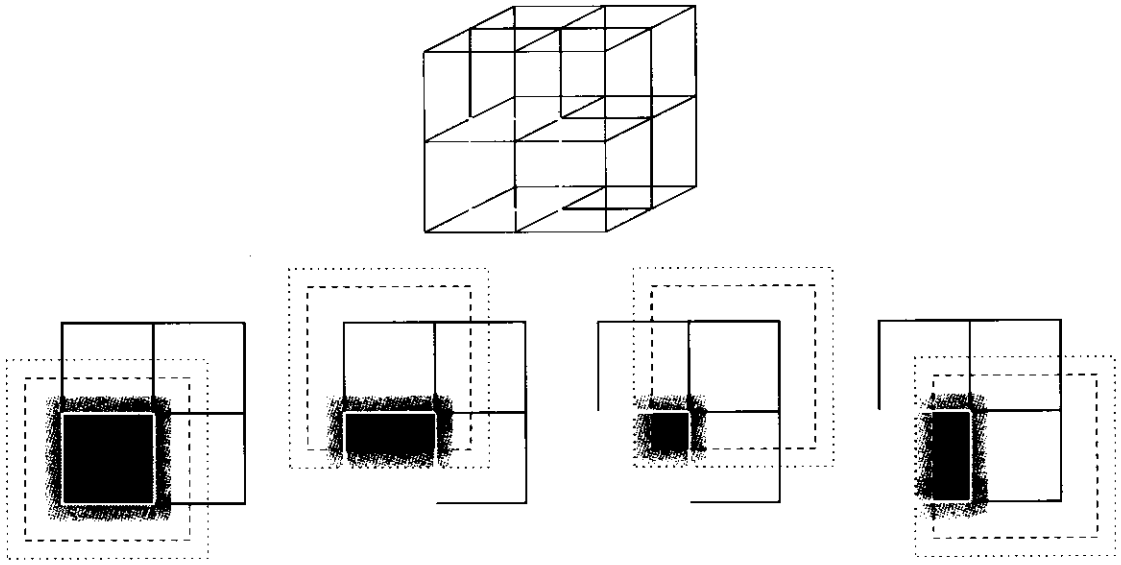

Figure 4: An illustration of setting boundary values for a final local solve in step 3 of MLC. For the face shown in red in the upper figure, in a layout in which there are eight cubes, the lower figures show the regions from which data are copied from faces of different neighboring boxes. Solid green lines indicate the boundaries of the boxes $\Omega^h_{k'}$. The dashed lines indicate the boundaries of the boxes $grow(\Omega^h_{k'}, s)$, and the dotted lines indicate the boundaries of the boxes $grow(\Omega^h_{k'}, s + Cb)$. Finer-grid data are copied to the red face from the nodes inside and on the edges of the regions shaded dark blue. Coarser-grid data are copied from nodes inside and on the edges of the regions shaded both dark and light blue, and then interpolated to nodes on the red face that are inside and on the edges of the regions shaded dark blue.

4 Performance Model

As mentioned previously, the principle behind Chombo-MLC is to trade off communication against computation. We next discuss these trade offs and show that they are reasonable. We describe a performance model, and use it to show that in theory the overheads are reasonable. In the following two sections we reconcile our predictions with practice.

In determining the computational overhead in Chombo-MLC, we will use the serial infinite-domain Poisson solver as a baseline. We will first show that the cost of our initial fine grid solutions is similar to the cost of a serial solution. The computational overhead in Chombo-MLC can be described as the sum of three costs: the extra computation required to calculate solutions on expanded local grids, the cost of the coarse grid solution, and the time required for the final local solutions on fine grid data. We will discuss each of these costs, and show that with the proper choice of a coarsening factor, Chombo-MLC should be able to scale to thousands of processors.

4.1 Serial Infinite-Domain Poisson Solver

Chombo-MLC reuses many of the same components as the serial infinite-domain Poisson solver. We will examine the computational costs involved in the serial solver first and compare this cost with the cost of the initial solutions in Chombo-MLC.

For simplicity, let us consider only cubical domains with edge length N. The operation counts for each step of the algorithm described in Section 3.1 are as follows.

1. Finding the solution to the Poisson equation on the inner grid using a fast (FFT) Poisson solver: $O(N^3 \log N)$.
2. Calculate a charge, q, along the inner grid boundary: $O(N^2)$.
3. Calculate boundary conditions at each point on the outer grid using FMM: $O(N^2)$
4. Find the solution to the Poisson equation on the outer grid using a fast (FFT) Poisson solver: $O(N^3 \log N)$.

Thus the serial infinite-domain solver operation count is bounded by the Dirichlet Poisson solve, and the overall computational cost of an infinite-domain solution is $O(N^3 \log N)$.

4.2 Practical Work Estimates

Since the computation required by the Poisson solvers used in our algorithm is nearly proportional to the number of points for which a solution is being found (ignoring the weaker $\log N$ term in the previous work estimates), we propose the following work estimates on the basis of these grid sizes. First, let us define W as an estimate of the work required for a Poisson solution with Dirichlet boundary conditions on a mesh Ω^h:

$$W = size(\Omega^h),$$

where the *size* operator returns the total number of points in the mesh Ω^h. Similarly, let us refer to W_k as the work required to compute a Dirichlet solution on a subdomain Ω_k^h.

Recall that the infinite-domain boundary calculation requires the solution of a Dirichlet problem on an enlarged domain, as defined by Equation 1 and shown by example in Table 1. After calculating the extents of $\Omega^{h,g}$ and $\Omega^{h,G}$ according to these requirements, we can define a work estimate fon an infinite-domain solution as

$$W^{id} = size(\Omega^{h,g}) + size(\Omega^{h,G}),$$

and also W_k^{id} as the corresponding estimate for the work required to compute a local initial fine grid solution within our MLC method.

Finally as an estimate of the total work per processor required for our MLC method, let

$$W_P^{mlc} = W_{coarse}^{id} + \sum_{k \text{ assigned to } P} (W_k^{id} + W_k)$$

where W_{coarse}^{id} an estimate of the work required to calculate the infinite-domain solution on the global coarse mesh. Note that to allow for the possibility of overdecomposition, multiple subdomains k may be assigned to a single processor P.

4.3 Minimizing the Cost of the Coarse Grid Solution

In comparing the computational cost of Chombo-MLC to a serial infinite-domain solver, the cost of computing the solution on the global coarse grid is overhead. We want to determine what range of problems can be solved with minimal computational overhead due to the coarse grid calculation. Our goal is to keep the cost of the coarse grid solution small enough that this overhead can be ignored.

As before, let N be the length of a side, thus N^3 is the total number of points. Let q be the number of subdomains on a side. Then q^3 is the total number of subdomains (the maximum number of processors) and

$$N_f = \frac{N}{q}$$

is the length of a local fine subdomain.

Let C be the coarsening factor, as defined previously, such that the size of coarse grid, N_c, is N/C. Since the coarse grid solution is not parallelized, we want $N_c < N_f$ in order to minimize the overhead due to the coarse grid. Thus we have

$$\frac{N}{C} < \frac{N}{q}$$

or

$$q < C.$$

4.4 Limits of Parallelism for the Method

As in most numerical libraries, an important consideration is how to optimize parameter settings that affect performance. The performance of Chombo-MLC is most affected by the choice of two parameters: q and C. However, various factors constrain the choice of these parameters, as well as the intrinsic parallelism, and these constraints limit performance.

When choosing the coarsening factor, C, we may affect both the size of the coarse grid solution and the size of the initial local solutions. Recall that our MLC algorithm requires us to find the initial local solutions on grids expanded by $2C$ in each direction. Thus increasing C may lead to extra work for the initial local solutions. As we have just seen in section 4.3, however, C needs to increase with q in order to keep the cost of the coarse solution in line with the local solutions.

Since the serial infinite-domain solver requires an annulus itself, there is a range of problems for which our algorithm is most suitable. We would like the coarsening factor for our MLC solver to be less than or equal to half the annulus size required by the infinite domain solver, i.e. $s_2/2$. The coarsening factor must also evenly divide the local grid size N_f. The maximum number of processors is then dependent on the choice of the ratio between q and C.

q/C	N_f	s_2	q	P	N^3
1/2	64	12	2	4	128^3
1/2	128	20	4	64	512^3
1/2	256	24	4	64	1024^3
1/2	512	44	8	512	4096^3
1	64	12	4	64	256^3
1	128	20	8	512	1024^3
1	256	24	8	512	2048^3
1	512	44	16	4096	8192^3
2	64	12	8	512	512^3
2	128	20	16	4096	2048^3
2	256	24	16	4096	4096^3
2	512	44	32	32768	16384^3

Table 2: Limits of parallelism for our MLC method with $q = C$. The number of processors, P, is taken to be the total number of subdomains, q^3.

Table 2 shows the limits of parallelism in terms of the maximum number of processors and maximum problem size for various local problem sizes, N_f, and ratios of q and C. If we assume, for instance that 128^3 problems will easily fit in local memory, users willing to expend a factor of two more computational effort can reach problem sizes of 1024^3 using 512 processors. Users willing to expend a factor of eight more computational effort could reach a problem size of 2048^3 on 4096 processors.

4.5 Future Improvements

The current implementation is limited by the relationship between the coarsening factor, C, and the number of subdomains per side, q. This restriction is due to computing the global coarse grid solution in serial. By parallelizing the global coarse solution we can vary C and q independently and extract significantly more parallelism from our MLC method. We have built a parallel implementation of the multipole calculation on the coarse grid infinite-domain solution and are considering alternatives for efficiently parallelizing the Dirichlet solves on the coarse grid while keeping communication requirements low. If our efforts are successful, Chombo-MLC could efficiently use thousands of processors without incurring additional computational overhead on the local solutions.

5 Results

In this section we present computational results which demonstrate the low communication overhead of Chombo-MLC on up to 512 processors. We also compare our performance results with the estimates presented earlier in Section 4

5.1 Hardware

We ran on NERSC's Seaborg IBM SP system, located at the National Energy Research Scientific Computing Center[1] Seaborg contains POWER3 SMP High Nodes interconnected with a "Colony"

[1] http://www.nersc.gov/nusers/resources/SP.

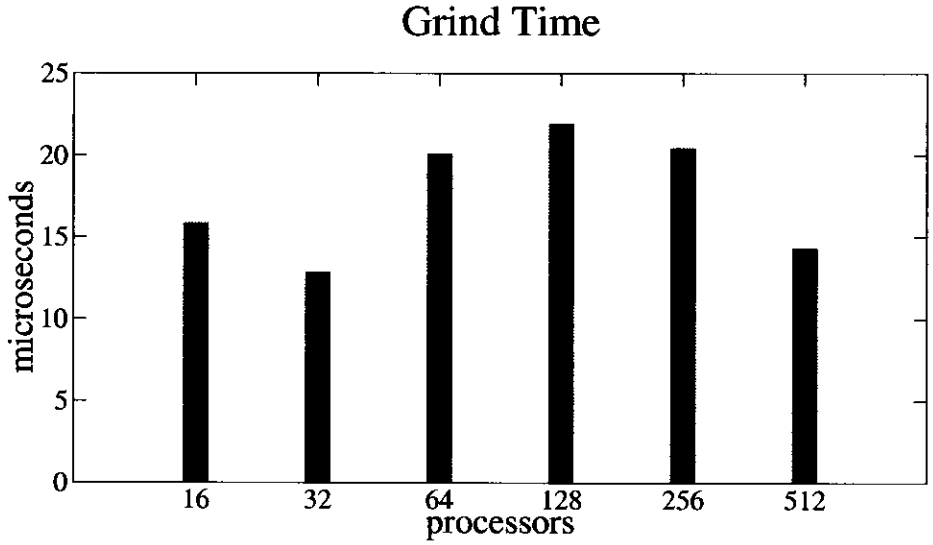

Figure 5: Grind time (processor-time per solution point) remains stable over a range of problem sizes.

switch. Each node is an 16-way Symmetric Multiprocessor (SMP) based on 375 MHz Power-3 processors[2], sharing between 16 and 64 Gigabytes of memory, and running AIX version 5.1.

Chombo-MLC is is written in a mixture of C++ and Fortran 77. We used the IBM C++ and Fortran 77 compilers, `mpCC` and `mpxlf`. C++ code was compiled with the IBM `mpCC` compiler, using options `-O2 -qarch=pwr3 -qtune=pwr3`. Fortran 77 was compiled with `mpxlf` with `-O2` optimization. We used the standard environment variable settings, and we collected timings in batch mode using *loadleveler*. The timings reported are based on wall-clock times, obtained with `MPI_Wtime()`. Each calculation was performed 3 times. Variations among the runs was less than 10%. The times reported are for the runs with the shortest total times.

Due to a memory bug in the current version of our code, some simulations failed to run in the memory available. In order to complete our simulations for this draft of our manuscript, we ran the 32 processor simulation on 4 nodes (using 8 processors per node rather than 16) and we ran the 512 processor simulation on 128 nodes (using 4 processors per node rather than 16). We are working on a fix to this bug and will have updated results in time for the camera-ready deadline.

5.2 Scalability

In order to measure performance, we scaled the work linearly with the number of processors. Ideally grind time, the processor-time required per solution point, would remain constant. The scaled speed-up tests shown in Figure 5 demonstrate that Chombo-MLC scales well up to 512 processors. The run parameters and timing results for the performance tests are shown in Table 3.

Communication overhead is relatively low in Chombo-MLC. As shown in Figure 6, communication overhead is less than 25% on up to 512 processors.

As can be seen in Table 3, time spent on the coarse grid solutions is approximately one third the time spent on fine grid solutions. Ideally, the time required for coarse grid solutions would be negligible, but these results match our expectations since we chose values of C closer to q than $2q$.

[2]http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/nighthawk.html

Input Parameters				Times for Each Stage (seconds)					Total	Grind
P	q	C	N	**Local**	**Red.**	**Global**	**Bnd.**	**Final**	(sec)	(μsec)
16	4	3	384^3	32.43	2.16	13.84	2.14	4.90	56.01	15.83
32	4	4	512^3	30.87	1.40	13.61	1.85	5.82	53.91	12.85
64	4	5	640^3	45.80	7.54	13.92	5.14	7.76	82.27	20.09
128	8	6	768^3	38.23	8.25	14.21	11.39	4.94	77.50	21.90
256	8	8	1024^3	45.89	6.73	14.06	10.78	6.02	85.73	20.44
512	8	10	1280^3	32.82	1.98	13.59	2.51	7.44	58.64	14.32

Table 3: Input parameters and timing breakdowns for runs. The Local and Global solutions require an infinite-domain solution, whereas the Final calculation solves a simpler Dirichlet problem. The time for Reduction (Red.) includes everything necessary to accumulate the coarsened local solutions into a single coarse grid for the Global solve. The time for Boundary (Bnd.) includes everything required to assemble correct boundary conditions for the calculation of the Final solution. P is the number of processors, q is the number of subdomains on a side, and C is the coarsening factor. Grind is the computation time per point, the grind time. Times may not sum exactly due to averaging.

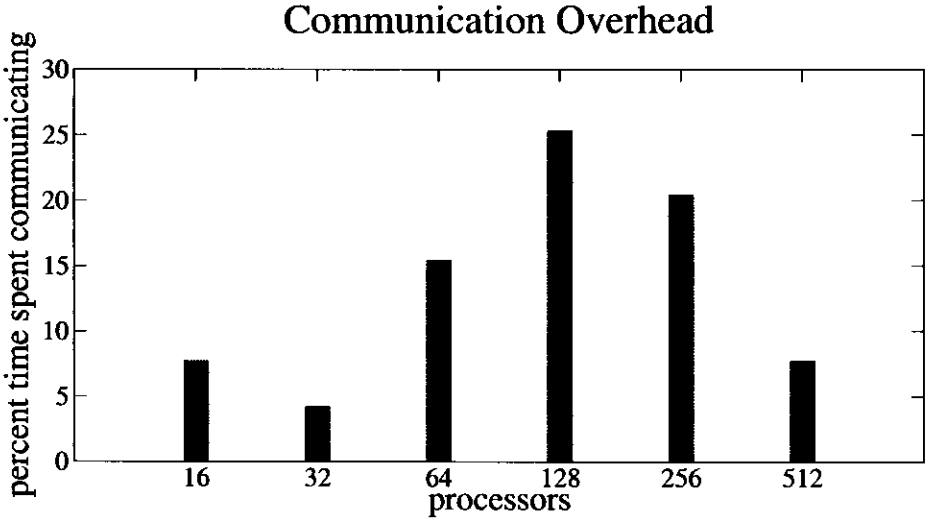

Figure 6: Communication overhead is small.

P	Time (sec)	W_k	Grind Time (μsec)
16	4.90	3.65×10^6	1.34
32	5.82	4.29×10^6	1.36
64	7.76	4.17×10^6	1.86
128	4.94	3.65×10^6	1.35
256	6.02	4.29×10^6	1.40
512	7.44	4.17×10^6	1.78

Table 4: Running times, points updated, and grind times for the final local solution phase of our MLC method.

P	Time (sec)	W_k	Grind Time (μsec)
16	4.90	13.06×10^6	2.48
32	5.82	13.95×10^6	2.21
64	7.76	13.30×10^6	3.44
128	4.94	13.06×10^6	2.93
256	6.02	13.95×10^6	3.29
512	7.44	13.30×10^6	2.47

Table 5: Running times, points updated, and grind times for the initial local solution phase of our MLC method.

In comparing our timing results to our work estimates, we start from the simplest building block and work up. In Table 4 we show the work estimates and running times for the final local solution phase, a simple Dirichlet Poisson solve, for each simulation which we ran. The grind time for these Dirichlet solutions ranges from 1.34 - 1.86 μsec and averages 1.52 μsec over the range of problem sizes. We believe the variation in performance is largely due to inefficiencies of the FFTW solver on meshes sizes of non-powers of 2.

Due to our choice of parameters, the global infinite domain solutions were performed on identical mesh sizes for all our problem sizes. Our work estimate for this stage of the computation, W^{id}_{coarse} is 7.07×10^6 mesh points. The running times for this phase of the computation were fairly consistent, and as a result, the grind times for this phase of the computation vary only modestly, from 1.92 - 2.01 μsec. Comparing these grind times with those for the Dirichlet solutions, we infer that the fast multipole method for the infinite-domain boundary calculation adds approximately 30% to the running time above what would be required for the two Dirichlet solutions. This is a significant improvement over our previous Scallop solver where the infinite-domain boundary calculation, performed by rather straightforward numerical integration, dominated the entire solution.

Grind times for the initial local solutions vary a good deal more than for the other phases, as shown in Table 5. The grind times calculated are also larger than those for the global infinite domain calculation. In part, this may be due to the extra work required during the infinite-domain boundary calculation to calculate the extra coarse grid values which are required later for interpolation.

We can also use the estimates of Section 4.2 to imagine the time required for an "ideal" infinite-domain solver. If we take the average of the grind times for the global infinite-domain solution (Table 4), 1.96 μsec, and apply that to the required number of point updates for the global fine-grid problem, we can estimate a lower bound on running times for infinite-domain calculations of

N^3	W/P	Ideal Time (sec)	Actual Time (sec)	Ratio
384^3	9.69	18.99	56.01	2.95
512^3	11.00	21.56	53.91	2.50
640^3	10.17	19.93	82.27	4.13
768^3	8.68	17.01	77.50	4.56
1024^3	9.71	19.03	85.73	4.51
1280^3	9.52	18.66	58.64	3.14

Table 6: Running times, points updated, and grind times for the initial local solution phase of our MLC method.

Code Version	Input Parameters				Times for Each Stage (seconds)					Total (sec)	Grind (μsec)
	P	q	C	N	Loc.	Red.	Glob.	Bnd.	Fin.		
Scallop	16	4	3	384^3	130.1	0.53	60.9	2.95	3.70	198.8	56.17
Scallop	128	8	6	768^3	187.7	1.89	67.3	6.42	4.42	270.7	76.49
Chombo	16	4	3	384^3	32.43	2.16	13.84	2.14	4.90	56.01	15.83
Chombo	128	8	6	768^3	38.23	8.25	14.21	11.39	4.94	77.50	21.90

Table 7: Comparison of running times of current Chombo-MLC and previous Scallop version.

various sizes. These estimates are compared to our actual running times in Table 6. The slowdown compared to an ideal solver ranges from roughly 2.5 to 4.6, trending only moderately higher with increasing numbers of processors.

In summary, we were are able to scale a problem up from 16 to 512 processors with, at worst, a factor of 1.7 increase in the grind time. Running times for each phase of our solver correlate fairly well with simple work estimates based on the number of points updated, with the global and local infinite-domain solutions taking slightly longer than the final Dirichlet solutions due to the extra work required for boundary condition calculations. Communication times are reasonably small for an elliptic partial differtial equation solver, generally staying well under 25% of the total running time.

5.3 Comparison to Previous Version

To our knowledge there are no other parallel finite-difference infinite-domain solvers with which we can compare our results. Our current results do show marked performance improvement over our previous Scallop implementation, however. The perfomance results from problem sizes for which we have direct comparisons are shown side by side in Table 7. The fast multipole method greatly reduces the cost of the infinite-domain boundary calculation in both the initial local solution and the global coarse grid solution. Communication times have increased somewhat but are still reasonable overall. The slightly greater communication times may indicate that further optimization is possible within the Chombo communication routines.

6 Conclusions and Future Work

We have presented a scalable 3D Poisson solver for free space problems that utilizes a method of local corrections that, in practice, eliminates communication overheads. The method employs a

philosophy for embracing technological change that substitutes relatively inexpensive computation for relatively expensive communication.

We described the design of the Chombo-MLC solver, which realizes our strategy. In practice, the performance of Chombo-MLC matches the expectations of our performance model quite well. Communication costs are less than one quarter of the total running time–generally significantly less–and total computation time is dominated by the time required for the initial fine grid calculations. The benefit of little communication comes at the expense of added computation, but this overhead is reasonable, and it remains almost constant, independent of the number of processors.

We are currently investigating ways to parallelize the global infinite-domain solution at the center of Chombo-MLC algorithm. Even modest parallelism in this phase of the computation would enable significantly increased parallelism overall allowing us to make use of thousands of processors. At the same time, parallelizing the global infinite-domain calculation would lift restrictions imposed on the initial local solutions, reducing further the computational overhead incurred by the method.

7 Acknowledgments

Peter McCorquodale and Phillip Colella are supported by the Mathematical, Information, and Computational Sciences Division of the Office of Science, U.S. Department of Energy under contract number DE-AC03-76SF00098. Greg Balls and Scott Baden were supported by the National Partnership for Advanced Computational Infrastructure (NPACI) under NSF contract ACI9619020. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. Chombo-MLC is publicly available at `http://www-cse.ucsd.edu/groups/-hpcl/scg/scallop/`.

References

[1] C. R. Anderson. A method of local corrections for computing the velocity field due to a distribution of vortex blobs. *Journal of Computational Physics*, 62:111–123, 1986.

[2] S. B. Baden and S. J. Fink. Communication overlap in multi-tier parallel algorithms. In *Proc. of SC '98*, Orlando, Florida, November 1998.

[3] S. B. Baden and S. J. Fink. A programming methodology for dual-tier multicomputers. *IEEE Trans. Software Engineering*, 26(3):212–26, March 2000.

[4] S. B. Baden and D. Shalit. Performance tradeoffs in multi-tier formulation of a finite difference method. In *Proc. 2001 International Conference on Computational Science*, San Francisco, CA, May 2001.

[5] G. T. Balls. *A Finite Difference Domain Decomposition Method Using Local Corrections for the Solution of Poisson's Equation.* PhD thesis, University of California, Berkeley, 1999.

[6] G. T. Balls, S. B. Baden, and P. Colella. Scallop: A highly scalable parallel poisson solver in three dimensions. In *Proc. SC '03*, Phoenix, AZ, November 2003.

[7] G. T. Balls and P. Colella. A finite difference domain decomposition method using local corrections for the solution of Poisson's equation. *Journal of Computational Physics*, 180(1):25–53, July 2002.

[8] R. Bank and M. Holst. A new paradigm for parallel adaptive meshing algorithms. *SIAM Journal on Scientific Computing*, 22(4):1411–1443, 2000.

[9] S. J. Fink, S. R. Kohn, and S. B. Baden. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1-2):61–82, April-May 1998.

[10] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *The Journal of Computational Physics*, 73:325–348, 1987.

[11] M. Holst. Applications of domain decomposition and partition of unity methods in physics and geometry. In I. Herrera, D. E. Keyes, O. B. Widlund, and R. Yates, editors, *Proceedings of the Fourteenth International Conference on Domain Decomposition Methods*, January 2002.

[12] R. A. James. The solution of Poisson's equation for isolated source distributions. *Journal of Computational Physics*, 25(2):71–93, October 1977.

[13] K. Lackner. Computation of ideal MHD equilibria. *Computer Physics Communications*, 12(1):33–44, 1976.

[14] B. F. Smith and O. B. Widlund. A domain decomposition algorithm using a heirarchical basis. *SIAM Journal on Scientific and Statistical Computing*, 11(6):1212–1220, November 1990.

[15] A. Sohn and R. Biswas. Communication studies of DMP and SMP machines. Technical Report NAS-97-004, NAS, 1997.

[16] A. K. Somani and A. M. Sansano. Minimizing overhead in parallel algorithms through overlapping communication/computation. Technical Report 97-8, ICASE, February 1997.